# Project User Programs Report

Group 43

| Name | Autograder Login | Email |
| --- | --- | --- |
| Sarvagya Somvanshi | student176 | sarvagya@berkeley.edu |
| Nathan Canera | student282 | nathancanera@berkeley.edu |
| John Chang | student271 | jchang04@berkeley.edu |
| Shengxiang Lin | student35 | linshengxiang@berkeley.edu |

# Changes

## Argument Passing

In my implementation of argument passing in the start_process function, I deviated from the design doc in a few ways to enhance the handling of arguments and stack alignment.

Instead of processing the arguments directly in the start_process function, I used strtok_r to parse the command-line arguments into an array (argv[]), storing each token while counting the number of arguments (argc).

My approach to stack alignment includes explicitly calculating the alignment offset using the formula `uint32_t align = ((uintptr_t)(if_.esp) - ((argc * 4) + 12)) % 16;` after pushing arguments onto the stack. Another thing we didn't mention was the actual structure of the stack so I went to the pintos documentation to figure out how to organize everything. What I ended up doing was putting the actual arguments on the stack, the padding, the pointers to the arguments, pointer to argv, pointer to argc, and the fake return address. The 16 byte alignment was supposed to be aligned to argc and wasn't affected by the fake return address.

I also structured the argument-passing process to account for edge cases like memory allocation failures when setting up the PCB and ensured modularization by

encapsulating the PCB logic separate from argument passing, resulting in better maintainability.

## Process Control Syscalls

I made a separate struct called `process_initializer` that takes in the parent's PCB and the command to be run by the child, which `process_execute` passes in as an argument to `start_process` instead of just passing in the command. I did this so that the child can modify the shared child struct in the parent's PCB.

Instead of directly storing the semaphore for loading the child process inside of the PCB, I made another struct called `loader` that stores both the semaphore and the success status of the child process. The `loader` struct is then stored in the PCB so that there is one place to both access the load semaphore and whether or not the child succeeded in loading.

I added an additional feature to the PCB called executable which is the file struct being executed by the process. I made the executable read only in `process_execute` by calling `file_deny_write` and reverted it to read and write in `process_exit` by calling `file_allow_write`. This way the running program / another program cannot modify an executable while it is being executed.

I tried implementing the reference counter and freeing the shared child struct after the reference counter equals 0, but I ran into a lot of errors that couldn't be fixed in time for submission. However, I now understand the importance of freeing process memory after it is no longer needed (since we didn't pass the OOM test) and will finish implementing this portion in Project 2.

## File Operation Syscalls

One of the key changes we made during syscalls in general was adding the `validate_ptr` and `validate_string` to handle invalid pointers robustly. We passed in the arguments from the syscalls operations to these validating functions to ensure that their pointers weren't invalid and were in the memory bounds required, a key point in operating systems in general.

Another key thing we ensured was to keep ensure that files were not modified during the `exec` function specifically, calling `file_deny_write()` when file was called and then `file_allow_write()` afterwards.

Lastly, we ensured we handled file descriptors table better, firstly initializing a blank `128-file sized` collection, dynamically mallocing the specific `file descriptors` when needed. Lastly, we ensured when `exit` was called to close all of the entries in the file descriptor table and freeing the table.

### Floating Point Operations

We forgot to mention that we would implement `sys_compute_e` in the design document so we implemented it by calling the built-in function in `float.c` but apart from that, we were faithful to our design in the design document.

---

# Reflection

This project was a great experience for all of us, concurrently learning and working on our ideas in operating systems. After Nathan took the lead on working on argument passing, we were able to distribute different tasks across the teams. John took the lead on process syscalls implementing `practice`, `halt`, `exit`, `exec` and `wait` protocols, Shengxiang took charge of floating point operations while Sarvagya and Nathan split the file syscalls between themselves, with Sarvagya implementing `create`, `open`, `write`, `filesize` and `remove`; while Nathan worked on `tell`, `seek` and `read`.

The work environment was amazing and all of us were ready to contribute in all the different ways possible. There were a lot of discussions on implementation styles and we were sure to clearly communicate what we working on, where we were stuck and explained the different concepts involved in our specific implementations so everyone in the group had a great idea of the implementation. This was crucial when debugging different errors as people were able to move vertically and help debug areas of programming they did not directly contribute to; being able to have a fresh perspective from your peers when in a frustrating bug was immense.

---

# Testing

**SEEK-TEST**

This test validates whether the file pointer is moved to the correct position and that operations after seeking are performed as expected. The test first creates a file and writes a fixed string, then file is reopened for reading and `seek()` is called to move the file pointer to different positions within the file. After each `seek()`, the test reads a portion of the file and checks if the read data corresponds to the correct location, verifying that the file pointer moved as expected.

The expected result is that the file pointer is correctly poved around and `read()` operations return expected data.
Expected Output:

```
(seek-test) begin
(seek-test) create "testfile.txt"
(seek-test) open "testfile.txt"
(seek-test) reopen "testfile.txt"
(seek-test) tell after seek to position 10
(seek-test) Read from position 10: "test "
(seek-test) compare read data at position 10
(seek-test) tell after seek to position 5
(seek-test) Read from position 5: "is a "
(seek-test) compare read data at position 5
(seek-test) tell after seek to end of file
(seek-test) close "testfile.txt"
(seek-test) end
seek-test: exit(0)
```

**Trivial Test Cases:**

1. **Incorrect File Pointer Update After `SYS_SEEK`:** If `SYS_SEEK` fails to update the file pointer properly, subsequent `read()` or `write()` operations will start from the wrong location. If the file pointer is not moved to position 10 as expected, the `read()` call will return incorrect data, such as starting the read from position 0 instead of 10.
2. **Seek Beyond File Boundaries:** If `SYS_SEEK` incorrectly handles positions beyond the end of the file, it may allow reading or writing outside the file's bounds. If the

kernel allowed seeking beyond the file's size without properly adjusting or restricting it, the test could either crash or read garbage data. For example, seeking to position 100 in a 20-byte file might lead to undefined behavior, and the test could produce invalid data output, causing a failure.

---

**Tell-Simple**

The test checks that the system correctly reports the file pointer's position after seeking and reading, ensuring alignment with the expected behavior.

The test starts by creating a file and writing data into it. Next it opens the file and verifies the initial file pointer position using `tell()`, which should be 0. After each `seek()` and `read()` operation, the test calls `tell()` to confirm the file pointer's updated position. The expected result is that `tell()` correctly returns the file pointer's position at each step, especially after seeking to various positions or reading a specific number of bytes.

**Expected**

```
(tell-simple) begin
(tell-simple) create "testfile.txt"
(tell-simple) open "testfile.txt"
(tell-simple) reopen "testfile.txt"
(tell-simple) tell at position 0
(tell-simple) tell after seek to position 10
(tell-simple) tell after read 5 bytes
(tell-simple) tell after seek to position 5
(tell-simple) tell at the end of the file
(tell-simple) close "testfile.txt"
(tell-simple) end
tell-simple: exit(0)
```

**Non-trivial Potential Kernel Bugs**

1. **Failure to Update File Pointer After `read()`**: If the file pointer is not updated after a `read()` operation, `tell()` will return the same value as before the read,

even though the file pointer should have moved forward by the number of bytes read. After reading 5 bytes from position 10, the file pointer should move to position 15. If the kernel doesn't update, the test would output:

```
tell-simple) tell after read 5 bytes: 10
```

instead of the correct value `15`.

2. **Incorrect Behavior of `SYS_TELL` After Seeking**If `SYS_TELL` does not return the correct file pointer position after a `seek()`, it may report an incorrect value, leading to misaligned reads or writes. After seeking to position 5, `tell()` should return 5. If the kernel doesn't properly track the file pointer, the test might output:

```
(tell-simple) tell after seek to position 5: 0
```

indicating the file pointer hasn't moved, causing the test to fail.

**Testing Experience:**
Writing tests for Pintos was an interesting challenge, especially because it pushed me to think about edge cases and potential kernel bugs in a much more detailed way. One thing I think could be improved is better debugging support—sometimes it's hard to pinpoint exactly where the issue lies in the kernel when a test fails. I learned a lot about how file systems track file pointers and how even small mistakes in pointer management can lead to significant bugs.